Efficient Embedded Software Design with Synchronous Models

M. Baleani[1], A. Ferrari[1], L. Mangeruca[1], A. Sangiovanni-Vincentelli[1,2]
[1]PARADES E.E.I.G., Via San Pantaleo 66, 00186 Rome, Italy
[1,2]Department of EECS, University of California, Berkeley CA 94709, USA
{mbaleani,aferrari,leon,alberto}@parades.rm.cnr.it

ABSTRACT

Model-based design is an important approach for embedded software. The method starts from a mathematical representation of the design problem and derives the software implementation from this representation. The model that has had most success especially for control dominated application is synchronous reactive. While this model simplifies the way of dealing with concurrency by decoupling functional and timing aspects, when implemented, it may be inefficient since the synchronous assumption implies constraints that are stronger than needed. We present in this paper a method for improving the efficiency of the software design process, by relaxing computation constraints, while preserving the synchronous computation semantics, with the introduction of a particular inter-task communication mechanism. We show how this mechanism can be implemented on single processor, multi processor and distributed implementation platforms.

Categories and Subject Descriptors: D.2 [Software]: Software Engineering

General Terms: Design, Theory.

Keywords: Synchrony, Model-based.

1. INTRODUCTION

Model based design is emerging as a solution to embedded software design issues. The tenet of this methodology is moving away from manual coding from informal specifications by capturing embedded software functional and non-functional requirements at the mathematical model level of abstraction. As the complexity of designs ramped up dealing with concurrency has become increasingly difficult, the interest in time-driven models has grown considerably also due to the availability of industrial tools and of a large body of theoretical results. Synchronous languages such as ESTEREL, LUSTRE and SIGNAL [2] and design environments like SIMULINK/ STATEFLOW are just a few examples of the so called *synchronous programming model* [2]. The synchronous assumption provides an effective and rigorous way of dealing with concurrency but poses strong constraints on the implementation.

The problem of implementing this paradigm efficiently on a single processor platform has been first discussed in [7] where the authors present an RTOS buffering support for inter-task communication that is able to preserve the synchronous semantics assuming 1) a fixed-priority deadline monotonic preemptive scheduler; 2) a task set with deadlines shorter than periods. In this paper, we propose a generalization of the communication mechanism proposed in [7]. Our starting point is not a particular implementation but rather an abstraction of a generic platform characterized by a timing model. A common timing model of implementation platforms in real-time embedded software is expressed in terms of periods, deadlines, response times and offsets of computation tasks. This allows us to deal with any task set and, most importantly, being independent of the implementation, to explore the design space effectively. Not only are we able to target single processor platforms with different scheduling policies, but also to address multi-processor and distributed architectures. Due to its generality, our approach has to confront issues that are not discussed in [7]. In particular, we focus on buffer insertion as a method to keep communication overhead to a minimum. Our approach to sizing the buffers is based on the timing model used to abstract the implementation platform and as such is capable of exploiting the characteristics of the entire design process from capture to implementation.

We realize that our approach to implementation optimization for synchronous language specifications could be viewed as a form of de-synchronization and as such this work can be related to the theoretical approaches proposed in [6], [3] and [4]. However, the analysis needed to relate all these approaches is beyond the scope of this paper and will be carried out in a forthcoming paper.

2. THE SPECIFICATION MODEL

Our analysis starts from a specification model where the system is described as a composition of mixed time- and event-triggered blocks that execute according to the synchronous assumption [2]. Such a model is well known in the literature, can be formalized in several ways [1] and is supported by several design tools, e.g. Simulink/Stateflow and SCADE. The synchronous assumption imposes a computation requirement on the final implementation, which can be captured as a deadline requirement of one "tick" (logical time) for computing all triggered blocks. Since the

EMSOFT'05, September 19–22, 2005, Jersey City, New Jersey, USA.
Copyright 2005 ACM 1-59593-091-4/05/0009 ...$5.00.

one tick deadline represents only a way of implementing the synchronous computation, more efficient implementations of the system can be obtained by removing such requirement. The work presented in this paper aims at increasing efficiency in the software implementation while preserving the synchronous computation semantics and can be viewed as a generalization of the work presented in [7].

Efficiency can be obtained by deferring the block's computation, as long as the deadlines coming from the system's timing specification are met, which however requires the introduction of buffers to store communication data and appropriate access mechanisms to retrieve the required data in the correct order. Here we have a trade-off, which in many practical cases pays off, because the computation burden is heavily reduced. The above relaxation can be regarded as a *desynchronization* of the system specification and clearly widens the implementation space. This relaxation can be explicitly represented in the model by introducing tagging functions as presented in the sequel.

By recalling that signals can be represented as functions of time, we can decompose them as follows. Let k denote the logical time. Without loss of generality k can be taken to be a variable over the set of natural numbers. Every signal in the system representation is a function $s(k)$ that returns the value of the signal at the logical time k.

Let us decompose the signal into two functions $s(k) = s'(T_s(k))$. We refer to the function $T_s(k)$ as the *tagging function* for s. The pair $(s(k), T_s(k))$ is usually referred to in the literature as an event on signal s and $T_s(k)$ is called its tag. The tagging function may be any function that composed with s' provides the original signal s, but for our purposes it is sufficient to select a function $T_s : \mathcal{N} \mapsto \mathcal{N}$, where $\mathcal{N}$ denotes the set of natural numbers.

Signals are put into relation by components. In particular, if P is the producer of signal s, then there is a relation between the tagging function of P's triggering signal and the tagging function of signal s. This is because P produces new outputs only when *triggering events* occur. The simplest way of stating this relationship is to let the two tagging functions be equal, i.e. $T_s(k) = T_{t_P}(k)$, where t_P denotes the triggering signal of P. As for $T_{t_P}(k)$, it may be defined, for example, as one returning the number of triggering events up to logical time k.

Consider a simple system where a producer P and a consumer C are interconnected by a signal s. When the consumer is triggered by a triggering event, say at logical time k, it must read the input value $s(k)$. The computation requirement can be relaxed if we provide C with the pair $(s(\cdot), k)$, where $s(\cdot)$ can be implemented as a buffer on s to store the values of the signal, while k is the index by which the buffer must be accessed. Now, we have seen above that $s(k) = s'(T_s(k)) = s'(T_{t_P}(k))$. Hence, the pair $(s'(\cdot), T_{t_P}(k))$ can be used in place of $(s(\cdot), k)$, which allows for a more convenient implementation because only a small subset of values need to be stored in the buffer.

From the discussion above it follows that correct implementations of the synchronous model can be obtained by implementing the following: 1) buffers on signals, 2) tagging functions and 3) *prod-cons precedence constraints.* The prod-cons precedence constraint is necessary to guarantee that when a consumer accesses the buffer with the proper tag, it will find valid data to read. In other words all data must be produced before being consumed.

3. SIZING COMMUNICATION BUFFERS

As introduced in the previous section, to relax the computation requirement of synchronous systems it is necessary to implement event buffering and tagging for every communication signal in the system specification. In the present section we provide a formula for sizing the communication buffers, based on the system's timing specification. In the sections to follow we will provide tagging techniques for different implementation platforms.

We take the timing model from the literature on real-time scheduling. Here, each system's component is associated with a time period T and a deadline D. The period can also be defined for sporadic event-based references as the minimum time separation between any two successive events. The deadline is the maximum time interval by which the k^{th} component's computation must be completed after the k^{th} triggering event. We also define a maximum and a minimum dynamic offset, denoted by O_{B_i,B_j} and o_{B_i,B_j}, respectively, of a component B_j with respect to a component B_i as the maximum and minimum time separation between a triggering event of B_j and the preceding triggering event of B_i. Note that the following inequalities hold for the dynamic offsets: $0 \leq o_{B_i,B_j} \leq O_{B_i,B_j} \leq T_{B_i}$.

A bound for the optimum buffer size is given by the minimum between the events written and those read on s within the maximum lifetime τ_s of events on s, i.e. the maximum time interval that elapses between the occurrence of an event and its consumption. In general, for one producer P and N consumers C_i on signal s, a bound for the overall buffer size is given by

$$\sum_{i=1}^{N} \min\left(\left\lceil \frac{\tau_{s,i}}{T_P} \right\rceil, \left\lceil \frac{\tau_{s,i}}{T_{C_i}} \right\rceil\right) \qquad (1)$$

Formula 1 gives a coarse bound for the multiple consumers case. The bound can be made tighter by observing that, no matter how many consumers read from s, the number of distinct events read cannot be larger than the number of events written to s by P. Consider the N consumers and let them be ordered by increasing lifetime, so that $\tau_{s,i} \leq \tau_{s,i+1}$. $\left\lceil \frac{\tau_{s,j}}{T_P} \right\rceil$ is an upper bound to the required buffer size for all consumers C_i with $i \leq j$, because $\tau_{s,j}$ is the longest lifetime for those consumers and $\left\lceil \frac{\tau_{s,j}}{T_P} \right\rceil$ provides the maximum number of events written by P within $\tau_{s,j}$. Hence, given $j = \max\left\{ i | \left\lceil \frac{\tau_{s,i}}{T_P} \right\rceil \leq \sum_{k=1}^{i} \left\lceil \frac{\tau_{s,k}}{T_{C_k}} \right\rceil \right\}$, a tighter bound on the buffer size is given by $\left\lceil \frac{\tau_{s,j}}{T_P} \right\rceil + \sum_{i=j+1}^{N} \left\lceil \frac{\tau_{s,i}}{T_{C_i}} \right\rceil$, where the first term represents a buffer shared among all consumers C_i such that $\tau_{s,i} \leq \tau_{s,j}$. Each addendum of the sum still accounts for a buffer dedicated to the i-th consumer C_i, as in formula 1, for those C_i such that $\tau_{s,i} > \tau_{s,j}$.

As for the longest lifetime on s for each consumer, observe that an event e_k produced by P on s will be read by a consumer C_i if there is a triggering event of C_i that occurs after e_k and before e_{k+1}, the next event produced by P. We call this condition an *interleaving* of events of P and C_i. Moreover, e_k will be alive until C_i completes its computation. Hence, the longest lifetime on s for C_i has to include the response time of C_i and the maximum offset of C_i with respect to P, i.e. $\tau_{s,i} \leq O_{P,C_i} + R_{C_i}$, where R_{C_i} is the maximum response time of C_i, i.e. the maximum time interval within which the computation of C_i is completed.

In all schedulable implementations, R_{C_i} is smaller than the deadline D_{C_i}. D_{C_i} can be used in place of R_{C_i}, when the response time is not available. The formula for the upper bound to the buffer size on signal s becomes:

$$\left\lceil \frac{O_{P,C_j} + R_{C_j}}{T_P} \right\rceil + \sum_{i=j+1}^{N} \left\lceil \frac{O_{P,C_i} + R_{C_i}}{T_{C_i}} \right\rceil \qquad (2)$$

When $D_i < T_i$, $\forall i$, the bound provided by formula 1 can be compared with the implementation presented in [7]. Note that this bound is never worse than the two-place buffer deployed in [7]. Rather, when $T_P + R_C < T_C$, one-place buffers suffice, since there is always at least an execution of P between two successive executions of C. The very rationale is that, formula 1 does not necessarily include a buffer location for the freshest value computed by the producer, because this is not always needed by the consumers. We can choose a looser bound to always guarantee a buffer location for writing the freshest value by observing that, under the condition $T_P \leq T_{C_i}$, we have $\left\lceil \frac{O_{P,C_i}+R_{C_i}}{T_{C_i}} \right\rceil \leq \left\lceil \frac{O_{P,C_i}}{T_P} + \frac{R_{C_i}}{T_{C_i}} \right\rceil \leq \left\lceil 1 + \frac{R_{C_i}}{T_{C_i}} \right\rceil = 1 + \left\lceil \frac{R_{C_i}}{T_{C_i}} \right\rceil$. The term $\left\lceil \frac{R_{C_i}}{T_{C_i}} \right\rceil$ accounts for all events that P can produce during the execution of C_i and 1 accounts for the reserved place to accommodate the freshest event.

Buffer optimization is more effective when we consider multiple readers on a single signal due to the introduction of the shared buffer accommodating all events produced by P over a given period of time. This optimization has not been considered in [7].

The bound provided by formula 2 may be further reduced by taking into account special but very common timing conditions, such as *time-disjoint* consumers such that $R_{C_i} \leq o_{C_i,C_j}$ and $R_{C_j} \leq o_{C_j,C_i}$, and *single-event* consumers for which $O_{C_i,C_j} \leq o_{P,C_j}$. In both cases C_i and C_j can share the same buffer locations, because in the former case the lifetimes of their readings do not overlap (disjoint computations), while in the latter the consumers read the same events. Hence, in formula 2 we do not need to consider both consumers, but only the one with the highest lifetime/period ratio, i.e. $\max\left(\frac{\tau_{s,i}}{T_{C_i}}, \frac{\tau_{s,j}}{T_{C_j}}\right)$.

4. IMPLEMENTATION OF SYNCHRONOUS SYSTEMS

In all software implementations, triggered blocks are mapped to tasks. When the computation carried out by a block B is required, the task B has been mapped to must be released by the scheduler. B's triggering events are thus associated to release requests of the corresponding task.

The access mechanism to the buffer must be strongly related with the tagging function in order to guarantee the correct computation. To simplify implementation, we will consider the looser bound guaranteeing a location for the freshest value, i.e. $\left\lceil \frac{O_{P,C_j}+R_{C_j}}{T_P} \right\rceil + \sum_{i=j+1}^{N} \left(\left\lceil \frac{R_{C_i}}{T_{C_i}} \right\rceil + 1 \right)$ Since the first term defines a buffer to store all the values written to s up to logical time k, the corresponding buffer can be accessed with the tag $T_{t_P}(k)$, defined as the number of triggering events to the producer up to logical time k modulo $\left\lceil \frac{O_{P,C_j}+R_{C_j}}{T_P} \right\rceil$, i.e. the size of the buffer. This requires computing $T_{t_P}(k)$ at each triggering event of the producer and consumers sharing the buffer. As for the second term, it represents $N - j$ independent buffers. Since the size of these buffers is tailored on the number of readings, their tagging functions depend on both $T_{t_P}(k)$ and $T_{t_{C_i}}(k)$. In this case, the tagging functions provide the number of interleavings between triggering events of P and C_i up to logical time k modulo $\left(\left\lceil \frac{R_{C_i}}{T_{C_i}} \right\rceil + 1 \right)$, i.e. the size of the buffer.

In the rest of the paper we will discuss how the mechanisms discussed above can be correctly implemented on most relevant architecture platforms.

4.1 Single processor implementations

In single processor architectures a unique scheduler sequentializes the computation of blocks' functions on the single execution resource. The tagging function of a signal is implemented by associating tagging operations to the producer and consumers of the signal. At each triggering event the appropriate tagging operations must be performed prior to the release of the corresponding triggered blocks.

The tagging function for the shared buffer provides the number of P's triggering events up to k, which is used to access the buffer. This can be more conveniently implemented using a pointer to the current buffer location that is being written by P. The buffer is handled as a circular queue and the pointer is incremented, modulo the size of the buffer, by P's tagging operation at each P's triggering event. Similarly, for consumers we need pointers that point to the locations that must be read. At any given consumer's triggering event, a pointer is created by the consumer's tagging operation and assigned the location currently pointed by the P's pointer. This ensures that when C_i executes it will read the correct value. C_i's pointer is no longer needed after C_i's completion.

The implementation of each of the $N - j$ independent buffers is the same as discussed above for the shared buffer, with one difference: the tagging function must count interleavings instead of writings on s. This affects the way the P's pointer is incremented. In particular, it must be incremented only if a consumer's pointer is moved to the current location, which indicates a new reading. To detect this condition we need a *read-flag* set by C_i's tagging operation before moving its pointer. When a triggering event of P occurs, the P's triggering operation tests the value of the read-flag and the P's pointer is incremented only if the read-flag is set. After incrementing the P's pointer, the read-flag is unset by P's triggering operation.

Tagging operations must all be atomic: they must run uninterruptedly and execute in the same order as triggering events. This can be guaranteed either by implementing tagging support directly in the RTOS as proposed in [7], or by demanding it to highest priority dedicated tasks that are released on triggering events.

The order P and C_i are executed does not affect the correctness of the implementation. Nonetheless, the prod-cons precedence constraint must be always satisfied. In single processor architectures, where a single execution resource is available, this order can be guaranteed by deploying either lock-free or locking methods. The former must guarantee that C_i's execution follows P's execution whenever C_i is triggered after P, because C_i needs the data produced by P. This can be obtained by statically scheduling C_i after P or by either static or dynamic priority assignment ensuring that $\text{Priority}_P \geq \text{Priority}_C$.

With locking methods, inversions in the execution order are possible and C_i may try reading a value P has not written yet. To block C_i's reading, a *valid-flag* can be added to each buffer location which is unset by P's tagging operation and is set by P when it completes writing the data. C_i must test the valid-flag before reading and is allowed to proceed only when the valid-flag is set.

4.2 Multi-processor implementations

In multi-processors architectures several parallel execution resources can be used to compute the overall system function and multiple RTOS instances schedule computation on each execution unit. This has potentially a strong impact on the capability of determining the total order among triggering events and on the atomicity of tagging operations. Since we are still considering a centralized architecture, a simple and efficient solution to the problem is to demand all tagging operations to a single execution unit. The computation of the system's block can still be allocated to different, parallel execution resources. It is mandatory that task executions follow the completion of tagging operations. This can be guaranteed by timing requirements or enforced, for example, by implementing all tasks as software tasks that are released only at the end of tagging operations. When P and C_i are assigned to different execution units, the prod-cons precedence constraint can only be guaranteed by a locking mechanism, such as for example the one described in Section 4.1.

4.3 Distributed implementations

Distributed implementations have the same problem of determining the total order among triggering events and ensuring the atomicity of tagging operations. Unlike multi processor architectures, however, it may be not viable to centralize all tagging operations due to higher communication cost (i.e. delay).

Since the case of P and C_i assigned to the same node is not different from single processor solutions, we focus our attention on the case when P and C_i are executed on different nodes, as their triggering functions are. Dedicated buffers are different to implement because of the atomicity constraint for the test and set operations on the read-flag. Hence, we can only implement a shared buffer for all consumers of a given signal and this must be local to the producer. To size the shared buffer we must use the formula $\left\lceil \frac{\tau_{s,N}}{T_P} \right\rceil$, where $\tau_{s,N}$ is the longest lifetime on s among all its consumers.

When C_i is triggered and its tagging operations are executed, C_i's pointers must be assigned the current value of the P's pointer, i.e. it must be $T_{t_{C_i}}(k) = T_{t_P}(k)$. However, communication delays may cause the value $T_{t_P}(k + D)$ to be assigned, instead. The problem can be solved by providing all nodes with a common time basis. Different techniques can be used. Independent, local clock sources can be deployed provided they feature a negligible drift within system life-time. Alternatively, we can resort to clock synchronization protocols like the one implemented in TTA [5]. Once a common time base is available, C_i can measure locally the value k_C of k when its triggering condition occurs. k_C can be fed to P, which can compute the corresponding tag $T_{t_P}(k_C)$ and finally provides C with the correct value $s'(T_{t_P}(k_C))$.

In terms of tagging operations, the time-stamps of P's triggering events are dynamically associated to the corresponding buffer locations by the P's tagging operation. When C requires a new data, it provides P with the time-stamp k_C of its triggering event. The tag for accessing the local buffer is computed as the least upper bound of P's time-stamps up to k_C and the correct data is retrieved and returned to C.

5. CONCLUSIONS AND FUTURE WORKS

In this paper we presented an approach for the efficient software implementation of synchronous systems. Efficient implementations require the introduction of buffers and tagging techniques. We presented formulas for sizing of the buffers driven by the system timing specification and we provided tagging techniques for different implementations, comprising single and multi processor solutions as well as distributed implementations.

We believe that this paper can be a solid basis for rigorous model-based embedded software design flows that start from synchronous specifications. The benefits of such design flows are two-fold: the implementation of the synchronous specification allows using simulation for early system verification, and the relaxation of the computation requirement at the specification level allows widening the implementation space for effective design space exploration.

Future works include an in-depth analysis of how the techniques proposed here relate to desynchronization theories proposed in the literature [4, 6], and how to use buffer sizes as a cost factor to drive the search for a feasible and cost-effective scheduling.

6. ACKNOWLEDGMENTS

This work has been partially supported by the European Community project IST-2-004527 (ARTIST-II), and by the NFS ITR CCR-0225610 (CHESS)[1].

7. REFERENCES

[1] Integrating model-based design and preemptive scheduling in mixed time and event-triggered systems. Technical Report TR-2004-12, Verimag Technical Report, 2004.

[2] A. Benveniste et al. Special section on another look at real-time programming. *Proceedings of the IEEE*, 79(9):1270–1336, sep 1991.

[3] A. Benveniste, B. Caillaud, L. Carloni, P. Caspi, and A. Sangivanni-Vincentelli. Heterogeneous reactive systems modeling: capturing causality and the correctness of loosely time-triggered architecture (LTTA). In *Proceeding of EMSOFT*, sep 2004.

[4] C. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli. The theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circtuits and Systems*, 20(9):1059–1076, sep 2001.

[5] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, jan 2003.

[6] D. Potop-Butucaru, B. Caillaud, and P. Le Guernic. Concurrency in synchronous systems. In *Proceeding of ACSD*, 2004.

[7] N. Scaife and P. Caspi. Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems. In *Proceedings of Euromicro Conference on Real-Time Systems (ECRTS'04)*, 2004.

[1] Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).